PacketLab: A Universal Measurement Endpoint Interface

Kirill Levchenko
UC San Diego
klevchen@cs.4ucsd.edu

Amogh Dhamdhere
CAIDA
amogh@caida.org

Bradley Huffaker
CAIDA
bhuffake@caida.org

kc claffy
CAIDA
kc@caida.org

Mark Allman
ICSI
mallman@icir.org

Vern Paxson
UC Berkeley/ICSI
vern@berkeley.edu

ABSTRACT

The right vantage point is critical to the success of any active measurement. However, most research groups cannot afford to design, deploy, and maintain their own network of measurement endpoints, and thus rely measurement infrastructure shared by others. Unfortunately, the mechanism by which we share access to measurement endpoints today is not frictionless; indeed, issues of compatibility, trust, and a lack of incentives get in the way of efficiently sharing measurement infrastructure.

We propose PacketLab, a universal measurement endpoint interface that lowers the barriers faced by experimenters and measurement endpoint operators. PacketLab is built on two key ideas: It moves the measurement logic out of the endpoint to a separate experiment control server, making each endpoint a lightweight packet source/sink. At the same time, it provides a way to delegate access to measurement endpoints while retaining fine-grained control over how one's endpoints are used by others, allowing research groups to share measurement infrastructure with each other with little overhead. By making the endpoint interface simple, we also make it easier to deploy measurement endpoints on any device anywhere, for any period of time the owner chooses. We offer PacketLab as a candidate measurement interface that can accommodate the research community's demand for future global-scale Internet measurement.

CCS CONCEPTS

• **Networks → Network measurement**;

KEYWORDS

Network measurement, PacketLab

ACM Reference Format:
Kirill Levchenko, Amogh Dhamdhere, Bradley Huffaker, kc claffy, Mark Allman, and Vern Paxson. 2017. PacketLab: A Universal Measurement Endpoint Interface. In *Proceedings of IMC '17, London, United Kingdom, November 1–3, 2017,* 7 pages.
https://doi.org/10.1145/3131365.3131396

IMC '17, November 1–3, 2017, London, United Kingdom

ACM ISBN 978-1-4503-5118-8/17/11…$15.00
https://doi.org/10.1145/3131365.3131396

1 INTRODUCTION

Having the right vantage points can make or break a network study. Whether it is observing Internet censorship, testing for network neutrality violations, or building a map of the Internet, researchers need access to end hosts from which they can conduct their measurements. Indeed, research groups invest considerable effort to secure access to such end hosts and operate them as measurement endpoints. The result has been a proliferation of Internet measurement platforms with different underlying architectures, implementations, functionalities, APIs, and user bases. Unfortunately, to run experiments on these platforms at scale, outside researchers and platform operators must overcome several obstacles:

- **Compatibility.** Each measurement platform has its own deployment and usage models. The experimenter must port her experiment to each platform individually—not an easy task. To break the N-interfaces-to-N-platforms paradigm, we need a single universal interface that works across all platforms, allowing experiments to scale easily.
- **Incentives.** Many platform operators provide technical support to the experimenter in the design phase of an experiment and operational support during deployment. The cost of providing these services falls on the operator and thus limits the number of outside experiments a platform can support. By reducing their support costs, platform operators can accept more outside experiments.
- **Trust.** On general-purpose platforms that do not limit experimenters to a fixed set of measurements, platform operators must trust each experimenter to behave according to a specified set of rules, and this can limit the set of experimenters admitted to a platform. To encourage platform operators to open their measurement infrastructure to a greater user base, operators need reliable mechanisms to guarantee compliance with their rules.

To lower these barriers, we propose a clean-slate measurement architecture we call PacketLab. PacketLab is *not* a new measurement platform; rather, PacketLab provides a lightweight, universal *interface* to existing measurement endpoints. Our value proposition to measurement platform operators is that PacketLab gives them control over how their platform is used and does so in a low-overhead way. By lowering barriers to sharing, PacketLab makes it easy to expose network vantage points to the measurement community, including new vantage points that are not part of an existing measurement platform. For experimenters, PacketLab provides a single interface to multiple measurement platforms, so that researchers

develop and test their experiments once and then run them on any endpoint exporting the PacketLab interface.

To meet these goals, PacketLab makes several unique design decisions. First and foremost, we argue that measurement endpoints should provide an *interface to the network*, and not to the endpoint computing hardware. Current platforms follow a model where adding a new experiment (whether one's own or external) requires updating software on the endpoint. Our philosophy is that adding a new experiment should require *no* changes to endpoints. This is only possible if we *decouple the platform from the experiment.* In PacketLab, measurement endpoints are simple packet senders and receivers. All experiment logic is located on a separate *experiment controller* that carries out the experiment. During an experiment, the experiment controller tells the endpoint what packets to send and what packets to capture and send back. This separation of experiment and platform means that measurement endpoints can be very simple while still supporting experiments of arbitrary complexity built on basic send and capture primitives. And because endpoints are simple, it is easy to add PacketLab support to existing measurement platforms. PacketLab experiment controllers are ephemeral, lasting only for the duration of an experiment. An experiment controller is provisioned and operated by the experimenter, not the platform operator, shifting costs typically borne by platform operators to the experimenter.

To control access to their endpoints, operators issue cryptographic certificates that authorize an experimenter to carry out a particular set of experiments. The experimenter's experiment controller then presents this certificate to each measurement endpoint she wishes to use. Certificates include restrictions that allow endpoint operators to limit the kinds of traffic that can be generated or collected at their measurement endpoints.

2 BACKGROUND AND RELATED WORK

End-host network measurement is an active area of research, with several measurement platforms in operation:

- RIPE Atlas [4]
- BISmark [30]
- FCC's MBA [2]
- CAIDA Ark [14]
- OONI [25]
- ICLab [1]
- Dasu [26]
- Netalyzr [19]
- MITATE [17]
- Scriptroute [29]
- PEERING [27]
- CAIDA Periscope [16]

We refer the reader to Bajpai and Schonwalder's recent survey [6] for an in-depth description of these efforts. Of particular relevance to this work are those platforms that explicitly invite outside experiments, namely BISmark, RIPE Atlas, FCC's Measuring Broadband America (MBA), Scriptroute, and Ark, as well as the more general PlanetLab [3] platform. Both BISmark and MBA started specifically for broadband speed measurements. CAIDA's Ark infrastructure was designed to be a community platform for active Internet measurement. RIPE Atlas was designed to support the operational needs of the RIPE community.

Ark, BISmark, and MBA can support arbitrary experiments that are vetted by the platform operator. Vetting as well as experiment development, testing, and deployment require involvement of the platform operator, and it is these costs that PacketLab aims to minimize. Furthermore, porting experiments to these platforms can be non-trivial. For example, when we wanted to extend our measurement of inter-domain congestion [20], originally written for Ark, to run on BISmark, differences between the platforms required us to re-design the experiment, moving most of the experiment logic off the endpoint. We argue that these platforms, and the experiments using them, stand to gain by offering an interface like PacketLab.

Scriptroute allows researchers to run scripts (written in the Ruby language) on the measurement endpoints without prior vetting, applying a local policy filter to limit the kind of traffic an endpoint can send. One of the unique features of Scriptroute is that it also allows measurement packet *destinations* to express a traffic policy by encoding it in a DNS record for the network.

In contrast, PacketLab moves all experiment logic off the endpoint, allowing researchers to write experiments in a language of their choice. PEERING, a measurement platform designed for routing experiments, follows the same philosophy, providing an OpenVPN tunnel to the researcher for handling traffic for an advertised route. To support general-purpose measurements, PacketLab endpoints also provide packet filtering, timestamping and scheduling primitives.

MITATE, a measurement platform aimed at mobile devices, keeps endpoints simple yet general by letting experimenters send an arbitrary, but pre-declared sequence of packets with a specific timing. The authors ensure that "these packets do not pose threats to other systems by matching them against signatures of known exploits using intrusion detection mechanisms."

At the more conservative end, RIPE Atlas supports a fixed (but useful) set of measurements that include ping, traceroute, DNS, SSL/TLS and some HTTP types. By limiting itself to measurements generally considered safe, RIPE Atlas achieves greater deployment (nearly 10,000 endpoints—an order of magnitude more than Ark and BISmark combined). RIPE Atlas underscores the importance of providing guaranteed limits to what an experiment can do in order to scale access to vantage points.

PacketLab is not another measurement platform, but an *interface* to existing and future platforms. In principle, each of the above platforms can provide a PacketLab interface in addition to their native interface. To make this possible, PacketLab will provide an access control and experiment monitoring system that allows each platform operator to enforce their desired experiment policy, while making as few assumptions as possible about the endpoint.

The measurement community has long sought a single interface to unify existing measurement platforms. There have been numerous discussions on federating existing measurement infrastructure, notably at the past three CAIDA AIMS workshops [11–13] (PacketLab itself was the subject of extended discussion at AIMS 2017). Projects such as MPlane [31] and Tophat [9] have attempted to either federate existing infrastructures, or build an intelligent *measurement plane* with probes that can execute measurements on-demand. Bajpai et al. [5] discussed the issue of platform integration at a Dagstuhl seminar, arguing that integration could be achieved by encouraging convergence towards an agreed-upon set of measurement primitives, tools, data storage formats, as well as a software management framework. PacketLab adresses the same need in a different way: moving all experiment logic to the experiment controller, leaving only the most simple packet send/receive mechanism on the endpoint.

3 ARCHITECTURE

The PacketLab architecture consists of *measurement endpoints*, *rendezvous servers*, and *experiment controllers*. Measurement endpoints may be software agents (e.g. Netalyzr), dedicated servers (e.g. Ark), or embedded systems connected to a home router (e.g. BISmark). Endpoints are managed and maintained by an *endpoint operator*. Except for granting permission to conduct an experiment, PacketLab does not specify how endpoint operators interact with measurement endpoints or if they do so at all. An endpoint operator can grant an *experimenter* (researcher) permission to use the endpoints he controls. The remainder of this section describes these elements of PacketLab in greater detail.

3.1 Measurement Endpoints

PacketLab measurement endpoints are software or hardware agents capable of sending and receiving packets on the Internet. A PacketLab endpoint provides an experimenter an interface to the network rather than an interface to the endpoint hardware. To run an experiment, an experiment controller operated by the experimenter interactively controls the measurement endpoint. In PacketLab terminology, *experiments* are short-lived interactive sessions between an experiment controller and an endpoint, generally lasting only a few minutes. A long-lived real-world research experiment will involve many short interactive sessions (PacketLab experiments). An endpoint's role during an experiment is simple: it sends packets that the experiment controller tells it to send, and it captures packets the experiment controller tells it to capture. All experiment logic is located on the experiment controller so that the measurement endpoint interface can remain simple and universal.

Network Primitives. The interface PacketLab endpoints export to experiment controllers is shown in Table 1. PacketLab endpoints can provide two kinds of access to the network: a raw IP interface, or a native TCP/UDP socket serviced by the endpoint's operating system. While the endpoint network access API (commands with an `n` prefix) resemble the BSD sockets interface, there are also important differences.

The first departure from BSD sockets is in how data is sent. To send data, the experiment controller uses the `nsend` command with a *time* parameter that tells the endpoint *when* it should send the data. This allows the experiment controller to schedule data to be sent at some future time, rather than immediately. (To send immediately, the controller specifies a time in the past.) The endpoint then attempts to send the data at the specified time, recording the time it was actually sent; an endpoint can retrieve this timestamp using the `mread` command described below. Delaying packets is useful when precise packet timing is necessary. For example, to measure bandwidth to a particular host, the experiment controller would schedule a sequence of packets to be sent a short time in the future. This avoids contention for the access link, since in most cases the same access link carries both PacketLab control and measurement traffic. By scheduling data to be sent later, rather than sending it immediately, traffic between the endpoint and experiment controller does not affect the bandwidth measurement.

The second difference between PacketLab and BSD socket interfaces is in how data is received. When an endpoint receives network data, it does not forward it to the experiment controller immediately, but buffers it internally until the experiment controller issues the `npoll` command. Only then does the endpoint sends all received data to the experiment controller. Buffering received data keeps the access link free of control traffic during a measurement, ensuring that PacketLab control traffic does not interfere with the experiment. If an experiment controller does not poll an endpoint quickly enough, an endpoint may run out of space to store all received data. When this happens, the endpoint simply stops reading (and buffering) experiment data. For TCP sockets, this will create flow control back pressure, while for UDP and raw IP sockets, the endpoint's host OS will simply drop packets. In addition to the received data, the `npoll` command also returns the number of packets and bytes dropped due to buffer exhaustion.

Opening a raw socket exposes the endpoint to all network traffic arriving on the endpoint's network interface. To limit which packets are returned, the experiment controller can install a packet filter using the `ncap` command. The *filt* parameter specifies the packet filter to use for filtering packets (see Section 3.4). The default behavior is to drop all packets, so an endpoint does not start capturing packets on a raw socket until the experiment controller installs a filter. The `ncap` command also takes a time parameter, which tells the endpoint when to stop capturing packets. This time can be arbitrarily far in the future, resulting in the filter remaining in place for the remainder of the experiment.

In raw mode, some incoming packets induced by the experiment may generate a response from the endpoint's host operating system. For example, an incoming TCP packet normally causes the operating system to send a RST packet if there is no matching TCP session. This can interfere with measurement experiments that create TCP sessions using the raw interface. To handle this, the packet filter installed by `ncap` specifies whether a packet should be ignored, consumed or mirrored to the experiment controller. (The mirror option is useful because it allows PacketLab to be used as a passive packet capture interface, for example, to capture packets at a network telescope [24].)

`nopen(sktid,proto)`
`nopen(sktid,proto,locport,remaddr,remport)`
The first form opens a raw IP socket on the endpoint. The second form opens a TCP or UDP socket with the specified local port to the specified remote address and port.

`nclose(sktid)` Closes the specified socket.
`nsend(sktid,time,data)` Queues data to be sent on a socket at a particular time.
`ncap(sktid,time,filt)` Installs a packet filter on a raw socket. Packets matched by filter will be captured until the specified time.
`npoll(time)` Polls endpoint for received network data, sending it to experiment controller. Waits until *time* if no data is available.
`mread(memaddr,bytecnt)` Reads *bytecnt* bytes starting from *memaddr* in endpoint virtual address space.
`mwrite(memaddr,data)` Writes *data* to location *memaddr* in endpoint virtual address space.

Table 1: Operations supported by PacketLab endpoints.

Not all endpoints may be able to support raw sockets. Many operating systems require superuser privileges to use raw sockets. If a PacketLab endpoint is a software agent running without root privileges, it will be unable to open a raw socket. (An experiment controller can determine if this is the case using the endpoint information commands, described next.) Endpoints that do not support the raw interface are still useful for experiments that only need a TCP or UDP socket, but not for experiments that need to create raw IP packets.

Endpoint Information and Configuration. An experiment controller may need additional information from the endpoint to carry out an experiment. For example, to craft a valid IP packet in raw mode, a controller needs to know the endpoint's internal IP address. (For endpoints behind a NAT, this address will be different from its external address.) A PacketLab endpoint makes this information such as its IP address, DHCP parameters, and the current socket state available to the controller via a structured block of memory that is accessed using the `mread` and `mwrite` commands listed in Table 1. The contents of this block of memory are also accessible to monitor programs (Section 3.4).

Timekeeping. The *time* parameter used in the `nsend` command and the timestamps on received packets are measured with respect to the endpoint's local clock. To keep endpoints as simple as possible, PacketLab does not require endpoints to keep accurate time. Instead, an endpoint makes its clock available as a read-only 64-bit value via the memory accessed using `mread` and `mwrite` commands. If an experiment requires accurate timing, the experiment controller should start by determine its clock offset with respect to the endpoint using a clock synchronization algorithm such as NTP [22]. By determining the clock offset of each endpoint, an experiment controller can then coordinate a multi-endpoint experiment that requires exact timing.

3.2 Rendezvous

Experiment controllers and measurement endpoints find each other with the help of a *rendezvous server*, which provides a publish-subscribe facility for experiment dissemination. Experimenters *publish* their experiments to a rendezvous server by sending the rendezvous server an *experiment descriptor*, which contains the address of the experiment controller, the experiment name, and a URL describing the experiment. When a PacketLab measurement endpoint starts up, it tries to find an experiment to run by contacting a rendezvous server and subscribing to a set of experiment *channels*. The rendezvous server sends the endpoint all experiment descriptors published to these channels. (We explain channels in Section 3.3). For each experiment descriptor it receives from the rendezvous server, an endpoint contacts the experiment controller given in the descriptor. The experiment controller can interact with the endpoint to determine whether it is suitable for a particular experiment based on its IP address and other information made available by the endpoint as described earlier.

An experiment descriptor does *not* contain the set of commands issued by the experiment controller, because experiments execute in an interactive fashion. An experiment controller is free to issue *any* commands during an experiment; the endpoint will use the monitor mechanism, described next, to determine whether it should execute each command.

Unlike experiment controllers, rendezvous servers are persistent. They constitute the only permanent infrastructure required by PacketLab. Their addresses may be hard-coded into the endpoint software like the names of DNS root servers. Rendezvous servers provide a simple service and are not themselves directly involved in experiments. We believe that two or three rendezvous servers can be maintained by the measurement community, just as NTP and PGP servers are managed by their respective communities.

3.3 Access Control

Access to rendezvous servers and measurement endpoints in PacketLab is controlled using cryptographic certificates similar to X.509 certificates used in the SSL/TLS ecosystem. Like X.509 certificates, PacketLab certificates may be chained to support hierarchical delegation. A certificate consists of a cryptographic hash of the signer public key, a cryptographic hash of the signed object, an optional list of restrictions, and a digital signature of the above.

There are two functionally different kinds of certificates: experiment certificates and delegation certificates. Both use the same format and differ only in the object being signed. In an experiment certificate, the object signed is an experiment descriptor (Section 3.2). In a delegation certificate, the object signed is another public key. A certificate may contain an optional list of *restrictions* on certificate applicability: validity period, experiment monitor (Section 3.4), buffer space limits, and priority (described later). The optional restrictions may apply to both kinds of certificates to limit the kinds of experiments an experimenter can run under those certificates.

Rendezvous Server Certificate Checking. The first use of certificates is to grant experimenters permission to publish experiments on a rendezvous server. To publish an experiment, an experimenter must first have a public/private key pair. The experimenter requests permission to publish her experiments from the rendezvous server operator. It is not the responsibility of the rendezvous server operator to act as an experiment gatekeeper, so permission to publish should be granted liberally. In practice, we expect the rendezvous server operator to delegate this task to a set of respected members of the community who can grant experimenters permission to publish experiments. The reason a certificate is required at all is to protect the rendezvous server against anonymous abuse by tying experiments to an individual or organization. The rendezvous server operator (or a delegate) grants a request to publish by signing a certificate where the subject is the hash of the public key of the experimenter. (Public keys are identified by their hash value.) An experimenter can then publish experiments on the rendezvous server by signing them with this key and providing the certificate issued by the rendezvous server operator to the rendezvous server. Each rendezvous server has a list of public keys whose signatures it accepts. An experiment descriptor must be signed (either directly or through delegation) by one of these keys for it to be accepted by the rendezvous server.

Measurement Endpoint Certificate Checking. Measurement endpoints follow a similar pattern. Each measurement endpoint has a set of public keys whose signatures it will accept. This set of trusted keys is installed and managed out-of-band by the endpoint operator. To run an experiment on an endpoint, an experiment controller must present the endpoint with an experiment descriptor

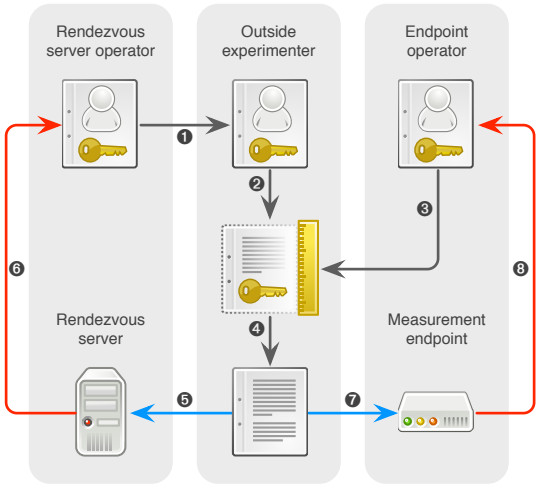

Figure 1: Authorization relationships in a PacketLab experiment. An experimenter obtains an experimenter certificate signed by a rendezvous server operator (❶). The experimenter then creates and signs a delegation certificate (❷) and has it signed by an endpoint operator whose endpoints she wants to use (❸). The delegation certificate allows the experimenter to create certificates for specific experiments (❹). Each experiment is published to a rendezvous server (❺), which accepts the experiment because the certificate chain establishes that the rendezvous server operator authorized the experimenter to publish experiments on the rendezvous server (❻). The experiment controller presents the certificate to each measurement endpoint (❼), which accepts the experiment because the certificate chain establishes that the endpoint operator authorized the experiment to run on the endpoint (❽).

that is directly or indirectly (via a chain of certificates) signed by one of its trusted keys. While an experimenter can ask the endpoint operator to sign an experiment descriptor for each experiment, it is more convenient to use delegation certificates. A delegation certificate signed by the endpoint operator authorizes a public-private key pair (controlled by an experimenter) to be used to sign experiment descriptors that will be accepted by the operator's endpoints. The experimenter must also sign the key pair corresponding to the delegation certificate so that a rendezvous server will accept experiments signed using that delegation certificate. Delegation can be extended several levels by forming a certificate chain. Figure 1 shows these authorization relationships.

Rendezvous Publish/Subscribe Channels. Recall that rendezvous servers provide a publish-subscribe system for experiment dissemination. Endpoints subscribe to a set of channels and receive all experiment descriptors published to those channels. The identifier used to describe a channel is simply the hash of a public key used to sign certificates. When a measurement endpoint connects to a rendezvous server, it subscribes to the set of channels corresponding to each of the public keys it trusts to sign experiment certificates. When an experimenter publishes an experiment to a rendezvous server, the experimenter includes the full certificate chain and corresponding public keys. This allows the rendezvous server to verify the certificate chain and broadcast the experiment to all endpoints that accept experiments signed by at least one of the keys in the certificate chain.

Contention. Access to endpoints is time-shared by multiple controllers. An endpoint can be involved in multiple concurrent experiments; however, at any given time, no more than one controller has control of an endpoint. If more than one controller wants to run an experiment on an endpoint, the endpoint decides which experiment to run based on its *priority*. When an experiment controller starts an experiment, it tells the endpoint at what priority it wants the experiment to run; this priority must not exceed the maximum priority specified in any certificate in the certificate chain used to authorize the experiment (see Section 3.3). If an experiment controller asks an endpoint to run a higher-priority experiment than what it is currently running, the endpoint notifies the experiment controller of the current experiment that its experiment has been interrupted, and then transfers control to the controller with the higher-priority experiment. The interrupted experiment is suspended until the higher-priority experiment completes or its controller suspends it by yielding control of the endpoint. The endpoint then returns control to the controller with the next highest priority suspended experiment. The ability to interrupt experiments ensures that low-priority experiments do not block high-priority ones. An endpoint operator can use this mechanism to grant outside researchers access to its endpoints with the understanding that their experiments may be interrupted at any time for the operator's own experiments.

As noted, unless interrupted by a higher-priority experiment, controllers have exclusive control of an endpoint during their experiment. This is necessary to prevent experiments from interfering with each other by competing for the same access link or endpoint buffers.

3.4 Experiment Monitor

In addition to the coarse certificate-based access control system described above, PacketLab experiment descriptors and certificates include a *monitor*. Monitors provide the mechanism by which an operator restricts what an experiment can do on an endpoint. An endpoint uses the monitor during the experiment to ensure that the experiment does not stray outside the behavior allowed by the endpoint operator.

Conceptually, a monitor is a black box that says whether an operation is allowed. Practically, a monitor is a program executing in a specialized virtual machine, a design borrowed from the BSD Packet Filter (BPF) [21], itself an evolution of earlier designs [10, 23]. In fact, BPF already nearly satisfies our requirements for a monitor mechanism. BPF has limitations, however. In particular, many implementations of BPF have a limited scratch memory of 16 32-bit words that does not persist across packets, making stateful filtering impossible. BPF programs must also be acyclic, a design that ensures that they execute in linear time (in the size of the program). Several packet filtering schemes attempt to overcome the limitations of BPF [7, 8, 15, 18, 28, 32, 33]; additional investigation is necessary to determine whether these, or a completely new scheme, would be most appropriate for PacketLab.

Recall that the `ncap` command included a packet filter argument that specifies which packets the controller wants the endpoint to capture. This packet filter is expressed as a program using the same mechanism as the monitor. Thus, both packet filters used with `ncap` and monitors attached to certificates determine which packets will be returned to the controller.

Memory. In addition to the information necessary to make an access control or packet filtering decision (e.g., packet data), a monitor program has access to auxiliary information about the endpoint and the state of the experiment via a structured memory block described earlier (Section 3.1). The monitor program sees this as a block of memory in its virtual address space. (Note that the address space seen by the monitor program and accessed by the controller using `mread` and `mwrite` is distinct from the virtual address of the host environment.) In addition, each monitor also has a block of private memory that persists for the duration of the experiment that is not accessible to the controller via the `mread` command.

High-level language for monitor programs. Writing a monitor in a (virtual) machine language is cumbersome. To make this task easier, we propose a simple C-like language we call Cpf that would be compiled to the representation interpreted by the endpoints. Cpf uses C syntax and semantics, but omits features like function pointers that are not necessary for creating monitor programs. We chose C because it is a familiar language to our target developer audience (network measurement experimenters), and, as such, would present no impediments to adoption. Furthermore, it allows us to directly use existing constant and structure definitions written in the C language.

A full discussion of Cpf is outside the scope of this article. However, Figure 2 shows how an endpoint operator might express a monitor for a traceroute experiment (we assume common header files such as `netinet/in.h` have been included, and that `union packet` is a union of structures containing common protocol headers). The endpoint operator would compile and attach this monitor to the experiment certificate it issues to an experimenter. The `send` entry point (invoked by the endpoint when the controller tries to send a packet) allows ICMP echo requests to any host. The program saves the destination in the `ping_dst` global variable. The `recv` function (invoked by the endpoint to determine whether a packet can be captured and passed on to the controller) allows ECHO replies from the destination and TIME EXCEEDED packets from any host. For the latter, the monitor ensures the source and destination of the returned IP header fragment match the original packet.

3.5 Limitations

Because PacketLab moves all experiment logic to the controller, any data to be sent by an endpoint during an experiment must first come from the controller. This means that there is necessarily a delay between when a controller commands the endpoint to send a packet and when the endpoint can actually send it. Experiments that require fast endpoint response times will be at a disadvantage, because the time between when an endpoint receives a packet and when it can generate a response that depends on the received packet will include the round-trip time between endpoint and controller. We note, however, that a round trip is only necessary if a sent packet depends on a received packet. If it does not, the controller can schedule the packet to be sent ahead of time. Timing measurements such as ping and traceroute are not affected by this, because what they need are precise timestamps (which PacketLab provides), rather than fast endpoint response times.

Another limitation of PacketLab is practical rather than technical. Most measurement platforms today follow the PlanetLab [3] model, where experiments run on the endpoint rather than on a separate controller. Developers will need to adjust to the PacketLab model, where, rather than sending packet directly, the programmer tells a remote endpoint to send a packet and may need to schedule packets in advance. We plan to develop libraries and VPN-style drivers to allow developers to code experiments to the old model but run them on PacketLab nodes.

4 PRELIMINARY RESULTS

We are in the early stages of prototyping PacketLab in order to validate the ideas presented in the paper. In the near future, we hope to provide the community with an open source endpoint reference implementation for evaluation. Our prototype supports a subset of operations shown in Table 1, namely commands to: open a TCP, UDP, or raw IP socket; send packets at a specified time; and capture and forward packets to the controller. Our endpoint does not yet support the rendezvous mechanism, certificates, or experiment monitors. Using our prototype endpoint, we implemented two experiments, described below.

Bandwidth measurement. To measure an endpoint's uplink bandwidth, we make it send a sequence of UDP packets to our server as quickly as possible, and then record the rate and which they arrive at the server. The controller first reads the current time t_0 on the endpoint (using the `mread` command). It then opens a UDP socket on the endpoint (using `nopen`) and schedules a block of UDP datagrams to be sent from the endpoint to the controller at time $t_0 + 5$ (using `nsend`). The controller then waits for the UDP packets from the endpoint, records their arrival times, and calculates the uplink bandwidth.

Traceroute. To reproduce the traceroute tool, an experiment controller creates a series of ICMP ECHO REQUEST packets with incrementing TTL values starting from 1 and the payload set to contain a two-byte sequence number. The controller first obtains the endpoint's current time t_0 as above, and then schedules the ICMP packets for transmission at some time $t_{\text{snd}} > t_0$. After scheduling the ICMP packets, the controller begins polling the endpoint for incoming packets, forwarding each to the controller with its receive timestamp (t_{rcv}). The sequence number is extracted from the packet and used to match the original ICMP's t_{snd} to calculate the round trip time as $t_{\text{rcv}} - t_{\text{snd}}$. Note that both timestamps are relative to the endpoint's clock. The controller sends packets to the endpoint until either an ICMP reply is received from the target destination or the next TTL value is greater then 40.

5 CONCLUSION

The aim of this work is to argue for a universal network measurement interface by presenting a particular design we call PacketLab. PacketLab gives endpoint operators a way to provide researchers access to measurement endpoints in a controlled manner. Endpoint operators can precisely enforce the kinds of experiments researchers can run on their endpoints using a mechanism based on packet filters. For experimenters, PacketLab is a uniform interface to all measurement endpoints supporting the PacketLab interface. Once an experimenter obtains a certificate (from an endpoint operator) granting her access to a set of endpoints, running the experiment does not require endpoint operator involvement, streamlining experiments and lowering support costs borne by operators of today's platforms.

```
in_addr_t ping_dst = 0; // destination of traceroute

uint32_t send(const union packet * pkt, uint32_t len) {
    if (pkt->ip.ver == 4 && pkt->ip.ihl == 5 &&
        pkt->ip.proto == IPPROTO_ICMP &&
        pkt->ip.src == info->addr.ip &&
        pkt->ip.icmp.type == ICMP_ECHO_REQUEST)
    {
        return len; // allow
        ping_dst = pkt->ip.dst;
    } else
    return 0;   // deny
}

uint32_t recv(const union packet * pkt, uint32_t len) {
    if (pkt->ip.ver == 4 && pkt->ip.ihl == 5 &&
        pkt->ip.proto == IPPROTO_ICMP && (
            (pkt->ip.icmp.type == ICMP_ECHO_REPLY &&
             pkt->ip.src == ping_dst) ||
            (pkt->ip.icmp.type == ICMP_TIME_EXCEEDED &&
             pkt->ip.icmp.orig.ip.src == info->addr.ip &&
             pkt->ip.icmp.orig.ip.dst == ping_dst)))
        return len; // allow
    else
        return 0;   // deny
}
```

Figure 2: Fragment of a monitor program for a traceroute experiment. The `send` entry point in the monitor is called by the endpoint to determine if a packet can be sent. The monitor first checks that the packet is an ICMP ECHO REQUEST packet and then stores the destination address in the global `ping_dst`. The `recv` entry point is called by the endpoint to determine whether the controller is allowed to capture the packet. It checks that the packet is an ICMP ECHO REPLY packet from the destination or a TIME EXCEEDED packet generated in response to the original ECHO REQUEST. Note that `recv` uses the global variable `ping_dst` to ensure that only packets corresponding to the original ECHO REQUEST are returned to the controller.

ACKNOWLEDGMENTS

The authors would like to acknowledge the participants of the 2017 AIMS workshop [13] for their insightful feedback on PacketLab. This work was supported by NSF grants CNS-1518918 and CNS-1513283, and by DHS S&T contract HHSP 233201600012C.

REFERENCES

[1] Internet Censorship Lab. http://www.internetcensorshiplab.com.
[2] Measuring Broadband America. https://www.fcc.gov/general/measuring-broadband-america.
[3] PlanetLab: An Open Platform for Developing, Deploying, and Accessing Planetary-scale Services. https://www.planet-lab.org.
[4] RIPE Atlas. https://atlas.ripe.net.
[5] V. Bajpai, A. W. Berger, P. Eardley, J. Ott, and J. Schönwälder. Global Measurements: Practice and Experience (Report on Dagstuhl Seminar #16012). *SIGCOMM Comput. Commun. Rev.*, 46(2):32–39, May 2016.
[6] V. Bajpai and J. Schonwalder. A Survey on Internet Performance Measurement Platforms and Related Standardization Efforts. *IEEE Communications Surveys and Tutorials*, 17(3):1313–1341, Apr 2015.
[7] A. Begel, S. McCanne, and S. L. Graham. BPF+: Exploiting Global Data-flow Optimization in a Generalized Packet Filter Architecture. In *Proceedings of the Conference on Applications, Technologies, Architectures, and Protocols for Computer Communication*, 1999.
[8] H. Bos, W. De Bruijn, M.-L. Cristea, T. Nguyen, and G. Portokalidis. FFPF: Fairly Fast Packet Filters. In *Proceedings of the USENIX Symposium on Operating Systems Design and Implementation*, pages 24–24, 2004.
[9] T. Bourgeau, J. Augé, and T. Friedman. TopHat: Supporting Experiments through Measurement Infrastructure Federation. In *in: Proceedings of the International Conference on Testbeds and Research Infrastructures for the Development of Networks and Communities (TridentCom*, 2010.
[10] R. T. Braden. A Pseudo-machine for Packet Monitoring and Statistics. In *Symposium Proceedings on Communications Architectures and Protocols*, pages 200–209, 1988.
[11] CAIDA. AIMS 2015: Workshop on Active Internet Measurements. https://www.caida.org/workshops/aims/1503.
[12] CAIDA. AIMS 2016: Workshop on Active Internet Measurements. https://www.caida.org/workshops/aims/1602.
[13] CAIDA. AIMS 2017: Workshop on Active Internet Measurements. https://www.caida.org/workshops/aims/1703.
[14] k. claffy, Y. Hyun, K. Keys, M. Fomenkov, and D. Krioukov. Internet Mapping: from Art to Science. In *IEEE DHS Cybersecurity Applications and Technologies Conference for Homeland Security (CATCH)*, pages 205–211, Watham, MA, Mar 2009.
[15] D. R. Engler and M. F. Kaashoek. DPF: Fast, Flexible Message Demultiplexing Using Dynamic Code Generation. In *Proceedings of the Conference on Applications, Technologies, Architectures, and Protocols for Computer Communication*, pages 53–59, 1996.
[16] V. Giotsas, A. Dhamdhere, and k. claffy. Periscope: Unifying Looking Glass Querying. In *Passive and Active Network Measurement Workshop (PAM)*, Mar 2016.
[17] U. Goel, A. Miyyapuram, M. Wittie, and Q. Yang. MITATE: Mobile Internet Testbed for Application Traffic Experimentation. In *Mobile and Ubiquitous Systems: Computing, Networking, and Services. 10th International Conference, MOBIQUITOUS 2013, Tokyo, Japan, Revised Selected Papers*, 2014.
[18] S. Ioannidis, K. G. Anagnostakis, J. Ioannidis, and A. D. Keromytis. xPF: Packet Filtering for Low-Cost Network Monitoring. In *Proceedings of the Workshop on High Performance Switching and Routing*, pages 116–120, 2002.
[19] C. Kreibich, N. Weaver, B. Nechaev, and V. Paxson. Netalyzr: Illuminating the Edge Network. In *Proceedings of the ACM Conference on Internet Measurement*, pages 246–259, 2010.
[20] M. Luckie, A. Dhamdhere, D. Clark, B. Huffaker, and K. Claffy. Challenges in Measuring Internet Interdomain Congestion. In *Proceedings of the ACM SIGCOMM Internet Measurement Conference (IMC)*, 2014.
[21] S. McCanne and V. Jacobson. The BSD Packet Filter: A New Architecture for User-level Packet Capture. In *Proceedings of the USENIX Conference*, 1993.
[22] D. L. Mills. Improved Algorithms for Synchronizing Computer Network Clocks. *IEEE/ACM Transactions on Networking*, 3(3):245–254, 1995.
[23] J. Mogul, R. Rashid, and M. Accetta. The Packer Filter: An Efficient Mechanism for User-level Network Code. In *Proceedings of the Eleventh ACM Symposium on Operating Systems Principles*, pages 39–51, 1987.
[24] D. Moore, C. Shannon, G. M. Voelker, and S. Savage. Network Telescopes. Technical Report 2004-04, Department of Computer Science and Engineering, University of California, San Diego, 2004.
[25] OONI. Open Observatory of Network Interference. https://ooni.torproject.org.
[26] M. A. Sánchez, J. S. Otto, Z. S. Bischof, D. R. Choffnes, F. E. Bustamante, B. Krishnamurthy, and W. Willinger. Dasu: Pushing Experiments to the Internet's Edge. In *Proceedings of the 10th USENIX Conference on Networked Systems Design and Implementation*, nsdi'13, pages 487–500, Berkeley, CA, USA, 2013. USENIX Association.
[27] B. Schlinker, K. Zarifis, I. Cunha, N. Feamster, and E. Katz-Bassett. PEERING: An AS for Us. In *Proceedings of the 13th ACM Workshop on Hot Topics in Networks*, 2014.
[28] J. Schulist, D. Borkmann, and A. Starovoitov. Linux Socket Filtering aka Berkeley Packet Filter (BPF). https://www.kernel.org/doc/Documentation/networking/filter.txt.
[29] N. Spring, D. Wetherall, and T. Anderson. Scriptroute: A Public Internet Measurement Facility. In *Proceedings of the 4th Conference on USENIX Symposium on Internet Technologies and Systems - Volume 4*, USITS'03, pages 17–17, Berkeley, CA, USA, 2003. USENIX Association.
[30] S. Sundaresan, S. Burnett, N. Feamster, and W. De Donato. BISmark: A Testbed for Deploying Measurements and Applications in Broadband Access Networks. In *2014 USENIX Conference on USENIX Annual Technical Conference (USENI X ATC 14)*, pages 383–394, 2014.
[31] The Mplane Consortium. MPlane: Building an Intelligent Measuremet Plane for the Internet. http://www.ict-mplane.eu.
[32] Z. Wu, M. Xie, and H. Wang. Swift: A Fast Dynamic Packet Filter. In *Proceedings of the USENIX Symposium on Networked Systems Design and Implementation*, pages 279–292, 2008.
[33] M. Yuhara, B. N. Bershad, C. Maeda, and J. E. B. Moss. Efficient Packet Demultiplexing for Multiple Endpoints and Large Messages. In *USENIX Winter Technical Conference Proceedings*, Jan. 1994.